Exhibit C

Information

Persistence Framework (Release 3.5)

Description

- Provide a set of APIs to support making domain objects persistent in a back-end datastore.

Services and Capabilities

- DatabaseWrapper – Allows access to database functionality. The DatabaseWrapper interface is used in the business component while the actual implementation type (ie. DatabaseWrapperTOPLinkImpl) is used in the factory methods.
- GetDatabaseWrapper – Allows access to the correct DatabaseWrapper implementation (ie. DatabaseWrapperTOPLinkImpl) based upon a key. The Initialization framework is used to associate datasource keys with their implementation types.
- DomainObjectFactory – Contains methods that allow for creation and querying of domain objects.
- PageEnumeration – A stateful session EJBean that allows more efficient access to large data result sets.
- Exceptions – Various exception classes handle error conditions that the framework creates.

Language

Java

Classes

Main Projects

Architecture (version 3.5)
Architecture TOPLink Dependants (version 3.5)

Main Classes

DatabaseWrapperBase
DatabaseWrapperTOPLinkImpl
DomainObjectFactory
DomainObjectFactoryTOPLinkImpl
PageEnumerationImpl

Implementation Packages

com.sprint.arch.persistence (version 3.5)

com.sprint.arch.persistence.TOPLink (version 3.5)

com.sprint.arch.persistence.weblogic (version 3.5)

com.sprint.arch.pageenumeration (version 3.5)

Implementation Interfaces

com.sprint.arch.persistence.DatabaseWrapper

- Contains method signatures of the base functionality that should be implemented in the database-specific implementations (ie. DatabaseWrapperTOPLinkImpl).

com.sprint.arch.pageenumeration.PageEnumeration

- Extends EJBObject. This is the EJB's remote interface.
- Contains method signatures of the base functionality of the page enumeration data structure.

com.sprint.arch.pageenumeration.PageEnumerationHome

- Extends EJBHome. This is the EJB's home interface.
- Contains method signatures for creation of a PageEnumeration EJB. Parameters include: page size, datasource name, factory interface class, and an array of handles.

com.sprint.arch.pageenumeration.PageableFactory

- Contains method signatures that a factory that wants to return PageEnumerations must implement.

Implementation Classes

com.sprint.arch.persistence.DatabaseWrapperBase

- Abstract class that contains fields and their accessors that each database wrapper implementation will inherit. These fields keep track of the datasource's name, Impl tag, package tag, and application property object.
- An additional method, getDomainObjectFactory(), returns an appropriate domain object factory object which is dependent upon the fields mentioned above.
- The getDatabaseWrapper() static method returns the appropriate DatabaseWrapper object based upon the type of datasource being accessed.

com.sprint.arch.persistence.DomainObjectFactory

- Abstract class that contains accessors which set the application properties and database wrapper fields.

com.sprint.arch.persistence.PersistenceException

- Extends Exception.
- Superclass of all exceptions thrown in the Persistence Framework.

com.sprint.arch.persistence.DatabaseException

- Extends PersistenceException.
- Thrown when there is an error in database communication.

com.sprint.arch.persistence.DatabaseWrapperException

- Extends PersistenceException.
- An exception class that is thrown when an exception occurs inside a database wrapper's methods.

com.sprint.arch.persistence.FactoryException

- Extends Exception.
- An exception class that is thrown when an exception occurs inside a domain object factory's methods.

com.sprint.arch.persistence.OptimisticLockException

- Extends PersistenceException
- Thrown when an optimistic lock fails.

com.sprint.arch.persistence.TOPLink.DatabaseWrapperTOPLinkImpl

- Extends DatabaseWrapperBase.
- Implements DatabaseWrapper interface.
- Contains additional methods that are TOPLink specific and can be used within the factory methods.

com.sprint.arch.persistence.TOPLink.DomainObjectFactoryTOPLinkImpl

- Extends DomainObjectFactory
- Abstract class that implements the accessors 'setDatabaseWrapper()' and 'getDatabaseWrapper()'. It has a field that holds onto the database wrapper object.

com.sprint.arch.persistence.TOPLink.TOPLinkConfigurator

- Contains methods that can be used to execute code before and after dabatase session login and loading of the TOPLink project.
- This class should be subclassed to meet application-specific needs.

com.sprint.arch.TOPLink.ServerSessionHolder

- Singleton class that holds onto TOPLink ServerSession objects that are established for each of the connected databases.

com.sprint.arch.persistence.TOPLink.TOPLinkConstants

- Contains common database initialization keys that are in the application's ini file.

com.sprint.arch.persistence.weblogic.ServerSessionThreadLoader

- This class spawns a thread that keeps a reference to the ServerSessionHolder singleton instance.
  com.sprint.arch.persistence.weblogic.ServerSessionThread

- This class keeps a reference to the ServerSessionHolder singleton instance so that it will not be garbage collected by the Java VM.

com.sprint.arch.pageenumeration.PageEnumerationImpl

- Implements SessionBean. This is the EJB's implementation.
- Contains methods to return the next and previous pages of a page enumeration.

Use of Other Frameworks/Components

If TOPLink is used then the appropriate TOPLink classes will be used. Otherwise, this framework is not coupled with any other frameworks.

Guidelines

Domain Object

Specification

The definition of a domain object is derived from analysis and design documents that detail the attributes and behavior for a specific domain object.

Interface

The first step in developing a domain object is to create its interface. The interface should define any behavior for the domain object as well as all public accessors. An interface for a **Customer** domain object might look like this:

```
public interface Customer {
        //Accessors for name
        public String getName();
        public void setName(String name);

        //Accessors for address
        public Address getAddress();
        public void setAddress(Address address);

        public boolean isValid();
}
```

Return types and parameter types in domain object interfaces will consist of basic types (String, int, boolean, etc.) and other domain object interface types (Address, Contact, etc.).

Interfaces keep business components and other code neutral to the implementation of domain objects. For example, a business component has visibility to the interface type Customer. A developer can create multiple implementations of Customer that implement the Customer interface; a developer might do this if a Customer can reside in a Versant database or in an Oracle database. Because the implementations

conform to the Customer interface, each implementation can be used wherever a Customer is needed. This makes the business components highly reusable; if the database of a domain object changes, the business component can just "plug in" the appropriate implementation for its domain objects.

Because we want to keep business components as reusable as possible, developers cannot directly create domain objects. The following is incorrect code:

```
public class BusinessComponent {
        public Customer createCustomer(String name, Address address) {
                Customer customer = new CustomerImpl(name, address);
                return customer;
        }
}
```

The correct method for creating domain objects is detailed in the section *Domain Object Factory*.

Implementation

The second step in creating a domain object is to implement it. The implementation of a domain object consists of a **class** definition that implements the domain object **interface**. The class name must be different from the interface name.

```
public class CustomerImpl implements Customer {
        private String name;
        private Address address;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public Address getAddress() {
                return address;
        }
        public void setAddress(Address address) {
                this.address = address;
        }
```

A domain object is implemented for each database in which its state resides. Therefore, if Customer state can reside in Versant and Oracle, a developer creates a Customer implementation for Versant and a separate implementation for Oracle. Domain object classes are named to reflect the database in which they live. A TOPLink (Oracle) Customer implementation would be defined as follows:

```
public class CustomerTOPLinkImpl implements Customer {
        protected String name;
        protected ValueHolderInterface address;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public Address getAddress() {
                //implemented later
        }
        public void setAddress(Address newAddress) {
                //implemented later
        }
        public ValueHolderInterface getAddressHolder() {
                return address;
        }
        public void setAddressHolder(ValueHolderInterface holder) {
                address = holder;
        }
        public void isValid() {
                //some validation code
        }
}
```

In this class definition, **CustomerTOPLinkImpl** is the name of the class. "Customer " has been appended with "TOPLinkImpl" to show that it is a **TOPLink** implementation for the Customer domain object. The class CustomerTOPLinkImpl implements the Customer interface and can be used wherever a Customer type is specified in code.

A business component has visibility to the domain object interface type, not the class type. The following code is incorrect:

```
public class BusinessComponent {
        public Customer createCustomer(args) {
                Customer customer = new CustomerTOPLinkImpl();
                return customer;
        }
}
```

The problem with the above code is that by calling **new** on the class **CustomerTOPLinkImpl**, the BusinessComponent class must be re-implemented if the database for Customer domain objects changes. The BusinessComponent would have the same code as above but call the constructor for another Customer implementation. Business components never use **new** directly. The correct method for creating domain objects is detailed in the section *Domain Object Factory*.

Since the implementation of a domain object is specific to a database, the domain object's fields can be defined using database types. In the case of Versant, the type Handle can be used for references to other domain

objects. In the case of TOPLink, the ValueHolder type can be used. The use of these types helps exploit efficiencies in the database implementation but adds constraints to field access.

Accessors

The following example illustrates the use of accessors on domain objects. We have the following interface:

```
public interface Customer {
        public Address getAddress();
        public void setAddress(Address address);
}
```

And the following class:

```
public class CustomerTOPLinkImpl implements Customer {
        protected String name;
        protected ValueHolderInterface address;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public Address getAddress() {
                return (Address) address.getValue();
        }
        public void setAddress(Address newAddress) {
                address.setValue(newAddress);
        }
        public ValueHolderInterface getAddressHolder() {
                return address;
        }
        public void setAddressHolder(ValueHolderInterface holder) {
                address = holder;
        }
        public void isValid() {
                //some validation code
        }
}
```

In this example, **Address** is an interface type.

There are two sets of accessors for the address field. One set is defined on the Customer **interface** and provides a view of the Customer's internal state consistent with a pure domain model. The other set is a private view for internal and infrastructure use.

In the example above, the first pair of accessors uses TOPLink commands to convert the ValueHolder into an Address and back to a ValueHolder. The second pair of accessors sets and gets a TOPLink ValueHolder.

Transient Domain Objects

Up to now, we have been discussing domain objects in the context of a database session. Typically, business component code follows the following steps: obtain a database wrapper, begin a session, obtain factories, create, find, and use domain objects, end the session, and return requested information to application controllers. The returned information *can* be in the form of domain objects.

Domain objects are considered **persistent** while in context of a database session and are not valid outside the database session. In the sequence above, the domain objects are considered persistent since they are created or found within the database session boundary. Since domain objects can be passed out of the business component, the business component must make them transient (not tied to the database session).

A transient domain object is a database neutral implementation of the domain object interface. For a Customer domain object, the developer creates a Customer interface, a CustomerTOPLinkImpl class (for TOPLink implementation), and a **CustomerTransientImpl** class. The **CustomerTransientImpl** class contains no database specific types or code.

A conversion method, `copyTransient()`, is defined on each domain object interface.

<<Note: copyTransient() was previously called externalClone(), and currently exists as such in the repository.>>

```
public interface Customer {
        Customer copyTransient();
}
```

The implementation of the Customer interface would then implement the method copyTransient().

```
//Database implementation
public class CustomerTOPLinkImpl implements Customer {
        protected String name;
        protected ValueHolderInterface address;
        Customer copyTransient() {
                Return new CustomerTransientImpl(name,
                        getAddress().copyTransient());
        }
        .....
}
//Transient implementation
public class CustomerTransientImpl implements Customer {
        protected String name;
        protected Address address;
        Customer copyTransient() {
                return this;
        }
}
```

In the database implementation above, `copyTransient()` constructs a transient customer with its fields. `copyTransient()` must be called on all fields that reference other domain objects. In the example above, the field

`address` references an Address object. Therefore, `copyTransient()` is sent to the address field.

Business component code looks like the following pseudo code.

```
public Customer getCustomer(String id) {
  // get the database wrapper from the DatbaseWrapperBase
  // appCtx is a member variable of the business component
  DatabaseWrapper dbWrapper =
      DatabaseWrapperBase.GetDatabaseWrapper(appCtx, "CustomerDB");

  // begin a session
  dbWrapper.beginSession();

  // get the customer factory
  CustomerFactory cf = (CustomerFactory)
      dbWrapper.getDomainObjectFactory(CustomerFactory.class);

  // get a customer from the factory using a find method
  Customer customer = cf.findById(id);

  //maybe do some stuff with customer

  // while still in the session, convert the persistent customer to
  // transient
  Customer transientCustomer = customer.transientCopy();

  // end the session
  DbWrapper.endSession();

  // return the transient customer
  return transientCustomer;
}
```

Domain Object Factory

Domain object factories create domain objects and retrieve them from the database. Domain object factories consist of an interface that specifies create and find methods available on the factory and an implementation specific to a database.

```
public interface CustomerFactory {
        public Customer create();
        public Customer findByName(String name);
        public Vector findAll();
}
public class CustomerFactoryTOPLinkImpl
  extends DomainObjectFactoryTOPLinkImpl
  implements CustomerFactory {
        public Customer create() {
                return new CustomerTOPLinkImpl();
        }
        public Customer findByName(String name) {
                //code to query TOPLink for a Customer by name
                return customer;
        }
        public Vector findAll() {
                //code to query TOPLink for all customers
```

```
            return customers;
        }
}
```

The factory maintains the boundary between implementation specifics and the business component. The business component gets a reference to a specific domain object factory and asks it to create or find the domain object for it. Factory methods return domain object interface types.

The factory is a convenient location for database queries. Any query that needs to be performed for a given domain object type is placed in a "find" method on the domain object factory implementation. The find method can make full use of database specific code to generate queries. Thus the reason for separating the factory interface from the implementation. The factory framework parallels the domain object framework. The developer implements a factory for each database containing its domain objects. The interface type allows the business component to interact with a single type while having the ability to "plug in" various implementations.

Since a factory implementation is specific to a given database, there must be a factory for the domain object factories. This is also parallel to the domain object framework. A business component cannot directly instantiate a domain object factory.

See the section *Database Wrapper* for more information on creating a factory.

Database Wrapper

The database wrapper provides an API for session and transaction management. The database wrapper is also the factory for domain object factories. To retrieve a domain object factory from the database wrapper, the developer simply calls a method on the database wrapper specifying the class name of the domain object factory to retrieve. The method returns type Object, so a developer must cast the return value to the specific factory type.

```
{
...
DatabaseWrapper dbWrapper = //get database wrapper using getDatabaseWrapper;
CustomerFactory customerFactory = (CustomerFactory)
    dbWrapper.getDomainObjectFactory(CustomerFactory.class);
...
}
```

Once obtained, the factory can be used to create and find domain objects. See the section *Domain Object Factory* for more information.

DatabaseWrapperBase

Currently, the DatabaseWrapperBase class has one static method: getDatabaseWrapper(). It takes a Properties object as its parameter and returns an appropriate DatabaseWrapper implementation object. The Configuration Framework is currently used to generate this Properties object, but it could also be created manually.

Getting DatabaseWrappers

To get a database wrapper, call the getDatabaseWrapper() static method on the DatabaseWrapperBase class. You must pass a valid Properties object that contains all of the necessary properties to instantiate a database wrapper including the logical database name (eg – CustomerDB or TollFreeRoutingDB). See the *Initialization Files* section for more information regarding the Properties object. The DatabaseWrapperBase uses the Properties object to determine which concrete class of DatabaseWrapper to create (ie – DatabaseWrapperTOPLinkImpl).

Getting DomainObjectFactories

To get a factory for a domain object, call the getDomainObjectFactory() method on the appropriate database wrapper. You can pass the fully qualified class name of the interface (eg – com.sprint.tollfree.factory.AddressFactory) as a String or a Class object.

The reference returned from the method call must then be cast to the appropriate Interface type (eg – AddressFactory).

VAJ Workspace

You will need the following projects to develop using the framework:

- Architecture Persistence(version 3.5)

If developing for TOPLink:

- Architecture Persistence TOPLink Dependants (version 3.5)
- TOPLink (v2.0.1 – 28/6/1999 – patch release)
- TOPLink Development Tools (v2.0.1 – 28/6/1999 – patch release)

Initialization

Initialization information is passed into the Persistence Framework through a java.util.Properties object. This object has various key/value pairs that affect the way the connection to the backend database is made. These key/value pairs are described in detail in the *Application-specific Initialization File Settings* section.

It is the responsibility of the party that is instantiating the database wrapper to create the appropriate Properties object. Use of the Configuration Framework along with the Application Environment Framework is suggested but not mandatory.

If your team chooses to use the Configuration and Application Environment Framework then your application's Property object will most likely contain prefixes on the key names to identify which datasource the keys are describing. For example, a key name of com.sprint.tollfree.customerDB.DatabasePassword would contain the password needed to access the customerDB datasource. This prefix must be "chopped off" before it is passed into the database wrapper. The getProperties(String prefix) method in the Application Environment Framework can be used for this purpose.

Further information on initialization can be found in documents on the Configuration Framework and Application Environment Framework.

Application-specific Initialization File Settings

Database Section

Each database has a section that describes the individual settings for the database. Below are the various properties that can be set. Properties are required unless otherwise noted in their descriptions.

DataSourceName – The logical datasource name for the database. Examples: customerDB, userDB, etc.

WrapperType – The fully-qualified DatabaseWrapper implementation class name. This is used by the getDatabaseWrapper() method to return the correct DatabaseWrapper implementation. Example: com.sprint.arch.persistence.TOPLink.DatabaseWrapperTOPLinkImpl.

DatabaseUserName – The login ID to be used by TOPLink when making a JDBC connection.

DatabasePassword – The password for the above login.

SourceType – Indicates whether the project information is found in a project file or a generated class; must be either *class* or *file*.

ProjectClass – If *SourceType* is *class*, this specifies the name of the class to load. The name must include the package name.

ProjectFile – If *SourceType* is *file*, this specifies the project filename.

ProjectPath – If *SourceType* is *file*, this specifies the path to the project file. It may not contain a trailing slash.

PackageTag – (Optional) If present, indicates the "tag" added to the package name of the interface class to yield the package name of the implementation class. The default for TOPLink is "TOPLink". If the interfaces and implementations are in the same package, then an empty string should be used (e.g. – 'PackageTag=').

ImplTag– (Optional) If present, indicates the "tag" added to the interface name to yield the name of the implementation class. The default for TOPLink is "TOPLinkImpl".

MaxConnections – (Optional) If present, specifies the maximum number of connections in the TOPLink ServerSession connection pool. If not specified, a default of 10 will be used.

MinConnections – (Optional) If present, specifies the minimum number of connections in the TOPLink ServerSession connection pool. If not specified, a default of 5 will be used.

MaxRetry – (Optional) If present, indicates the maximum number of times to try to establish a database connection in the event that the initial attempt fails.

Driver – (Optional) If present, indicates the JDBC driver class to use in place of the one specified in the TOPLink project.

ConnectionString – (Optional) If present, indicates the database connection URL to use in place of the one specified in the TOPLink project. This can be used to connect to a database session pools.

ConfigClass – (Optional) This sets the class that will be used in place of the standard TOPLinkConfigurator which can be used to execute code before and after dabatase session login and loading of the TOPLink project. Should subclass com.sprint.arch.persistence.TOPLink.TOPLinkConfigurator

EnableSQLTrace – (Optional) Enables and disables SQL trace output from TOPLink. Must be either *true* or *false*.

Optimize – (Optional) Enables data conversion optimization. This has been known to cause problems with Date objects. Must be either *true* or *false*.

EJB Section

This section of the ini file contains mappings of an EJB's home interface class name to its JNDI name.

Initialization Sub-section:

java.naming.factory.initial – Indicates the factory used to produce InitialContext objects.

java.naming.provider.url – Indicates the provider URL of the Naming Service provider.

Currently there are two settings in use, depending on what environment you are using.

WebSphere Test Environment (i.e. – inside of VAJ)

java.naming.factory.initial=com.ibm.jndi.CosNaming.CNInitialContextFactory

java.naming.provider.url=iiop:///

WebLogic Application Server

java.naming.factory.initial=weblogic.jndi.TengahInitialContextFactory

java.naming.provider.url=t3://localhost:7001

Mappings:

HomeInterfaceClass=JNDI Name – For each EJB in the application, there must be an entry which maps its home interface class name (including package name) to the name registered with JNDI.

Examples:

com.sprint.arch.pageenumeration.PageEnumerationHome=PageEnumerationHome

prototype.businesscomponent.CustomerComponentHome=CustomerComponentHome

Java Persistence Framework Release 3.5

Technical Reference Guide

Example INI File ("#" indicates a comment)

```
#-----------
#CustomerDB
#-----------
com.sprint.tollfree.customerDB.DataSourceName=custDB
com.sprint.tollfree.customerDB.WrapperType=
com.sprint.tollfree.com.sprint.arch.persistence.TOPLink.DatabaseWrapperTOPLinkImpl
com.sprint.tollfree.customerDB.DatabaseUser=db_owner
com.sprint.tollfree.customerDB.DatabasePassword=db_owner

# com.sprint.tollfree.customerDB.PackageTag=
# com.sprint.tollfree.customerDB.ImplTag=

com.sprint.tollfree.customerDB.SourceType=class
com.sprint.tollfree.customerDB.ProjectClass=com.sprint.tollfree.TFProject
com.sprint.tollfree.customerDB.EnableSQLTrace=true
com.sprint.tollfree.customerDB.ConfigClass=com.sprint.tollfree.domain.TOPLink.PVCConfigurator

# com.sprint.tollfree.customerDB.Driver=
# com.sprint.tollfree.customerDB.ConnectionString=
# com.sprint.tollfree.customerDB.ProjectFile=
# com.sprint.tollfree.customerDB.ProjectPath
# com.sprint.tollfree.customerDB.Optimize=false
# com.sprint.tollfree.customerDB.MaxRetry=

#-----
#EJB
#-----
#For WebSphere use:
#java.naming.factory.initial=com.ibm.jndi.CosNaming.CNInitialContextFactory
#java.naming.provider.url=iiop:///

#For WebLogic use:
java.naming.factory.initial=weblogic.jndi.TengahInitialContextFactory
java.naming.provider.url=t3://localhost:7001

com.sprint.arch.pageenumeration.PageEnumeration=PageEnumerationHome
com.sprint.ion.components.VirtualCircuitOrderComponent=VirtualCircuitOrderComponentHome
com.sprint.ion.controllers.VirtualCircuitOrderController=VirtualCircuitOrderControllerHome
#-----------END-OF-INI-----------
```

TOPLink-Specific Considerations

Approach

The Persistence Framework TOPLink implementation (release 3.12 and later) is now making use of a Server-Client model in relation to database sessions. See figure 1 below for a graphical view of the architecture.

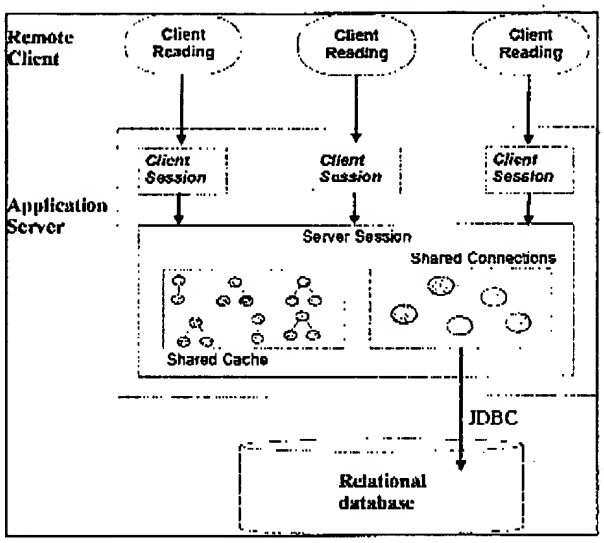

Figure 1. Multiple client sessions reading the database using the server session.

Benefits gained from using the Server Session architecture include one time loading of TOPLink project information, connection pooling, and a shared cache.

Based upon this implementation, the Server Session needs to be running before any client makes a database session request. The Server Session is responsible for loading the project information and creating a pool of database connections to service clients. This change does not impact the application code but it will change the ini file and require that the server session be somehow started before any client requests. To start the server session, code needs to be added to the application's initialization method to start a server session for each datasource. This is discussed in greater detail in the *Setup* section below. To ensure that a ServerSession is always available for any client that wishes to access it, we have taken the approach to store the ServerSession in the hashtable of a "well known" singleton class. To keep the singleton class from being garbage collected or unloaded for the lifetime of the application server another class is provided that starts a thread class. The thread class keeps a reference to the singleton class as a local variable in its run method. It then sleeps for the defined amount of time.

Setup

Additional code will need to be added to each application's initialization method. The code starts the necessary server sessions for each datasource that the application will access. An example might look like this:

```
//The datasource name is set. This is different for each datasource.
String dataSource = new String("customerDB");

//Get a "chopped" Properties object from Application Environment Manager
Properties dbInfo =
  ApplicationEnvironmentManager.instance().get(applicationID).getProperties(dataSource);

//Get a reference to the ServerSessionHolder and create the server session.
ServerSessionHolder holder = ServerSessionHolder.instance();
holder.createServerSession(dbInfo, dataSource);
```

This is repeated for each individual datasource.

The MaxConnections and MinConnections initialization file keys (mentioned in the *Application-specific Initialization File Settings* section) are used to control the ServerSession. MaxConnections specifies the maximum number of connections in the connection pool, while MinConnections specifies the initial number of connections. The default connection pool has minimum of five connections and a maximum of 10 connections.

Running Within the WebLogic Application Server

Make sure that the com.sprint.arch.persistence.weblogic package has been exported to your WebLogic server environment (a good choice is to put it in your <weblogic root>/classes directory). You will find this package in the Architecture WebLogic Dependents project.

Add the following lines to your weblogic.properties file:

```
weblogic.system.startupClass.ServerSessionStarter= \
      com.sprint.arch.persistence.weblogic.ServerSessionThreadLoader
weblogic.system.startupArgs.ServerSessionStarter=interval=60
```

This startup class ensures that the singleton class is not garbage collected or unloaded while the weblogic server is running. The first line tells the server to create an instance of ServerSessionThreadLoader class and invoke its startup() method when the server is started. The second line specifies a key-value pair where the key is interval and the value is the interval the thread should sleep for.

Not running within the WebLogic Application Server

If it's your bean that needs access to the server session, use the Initlalization Bean in the Architecture VisualAge Dependant Package. Add the code from the *Setup* section above to start the server session.

If your test client needs access to the server session, then add the code in the *Setup* section above to your client.

Page Enumeration

Description

The page enumeration is a data structure that allows a developer to "page" forward and backward through a large collection of domain objects. Typically, an application controller will ask a business component for a collection of domain objects. The business component delegates this request to a domain object factory. The domain object factory queries the database and creates a page enumeration that eventually gets passed back to the application controller. The page enumeration is meant for use outside a business component.

Creation

Page enumerations are created in domain object factories. The domain object factory developer writes a find method on the factory whose return value is a page enumeration. The find method queries the database for domain objects but only retrieves the handles for the objects. **Handles** are key data that can be used to get full domain objects from the database. In Versant, the query returns an enumeration of Handles. In TOPLink (Oracle), the developer must construct a query that returns key information (such as identifiers) and not full domain objects. Using handles to create a page enumeration is more efficient than using full domain objects.

Design

Interfaces/Public API

```
Package com.sprint.arch.persistence; (Version 3.5)
public interface DatabaseWrapper {
        public void beginSession() throws DatabaseWrapperException, DatabaseException;
        public void beginTransaction();
        public void commit()
                throws DatabaseException, OptimisticLockException,
                DatabaseWrapperException;
        public void delete(Object domainObject) throws DatabaseWrapperException;
        public void delete(Vector domainObjects) throws DatabaseWrapperException;
        public void endSession();
        public String getDataSourceName();
        public DomainObjectFactory getDomainObjectFactory(Class interfaceName)
                throws FactoryException;
        <<Deprecated>> public DomainObjectFactory getDomainObjectFactory(String name)
                throws FactoryException;
        public boolean inSession();
        public boolean inTransaction();
        public Object makePersistent(Object domainObject)
                throws DatabaseWrapperException;
        public Vector makePersistent(Vector domainObjects)
                throws DatabaseWrapperException;
```

```
            public void rollback() throws DatabaseWrapperException;
}
public abstract class DatabaseWrapperBase {
            public DatabaseWrapperBase(Properties appProps, String dataSourceName,
                        String factoryImplTag, String factoryPkgTag);
            final protected Properties getApplicationProperties();
            public static DatabaseWrapper getDatabaseWrapper(Properties appProps)
                        throws FactoryException;
            <<Deprecated>> public static DatabaseWrapper
                        GetDatabaseWrapper(Properties appProps) throws FactoryException;
            public String getDataSourceName();
            protected DomainObjectFactory getDomainObjectFactory(Class factoryInterface,
                        DatabaseWrapper dbWrapper) throws FactoryException;
            final protected void setApplicationProperties(Properties appProps);
}
public abstract class DomainObjectFactory implements Serializable {
            protected final Properties getApplicationProperties();
            protected final void setApplicationProperties(Properties appProps);
            protected abstract void setDatabaseWrapper(DatabaseWrapper databaseWrapper);
}
public class PersistenceException extends Exception {
            public PersistenceException();
            public PersistenceException(String s);
            public PersistenceException(String s, Throwable ex);
}
public class DatabaseException extends PersistenceException {
            public DatabaseException();
            public DatabaseException(String s);
            public DatabaseException(String s, Throwable ex);
}
final public class DatabaseWrapperException extends PersistenceException {
            public DatabaseWrapperException();
            public DatabaseWrapperException(String s);
            public DatabaseWrapperException(String s, Throwable ex);
}
public class OptimisticLockException extends PersistenceException {
            public OptimisticLockException();
            public OptimisticLockException(String s);
            public OptimisticLockException(String msg, Throwable rootCause);
}
public class FactoryException extends Exception {
            public FactoryException();
            public FactoryException(String s);
            public FactoryException(String s, Throwable ex);
}

package com.sprint.arch.persistence.TOPLink; (Version 3.5)
final public class DatabaseWrapperTOPLinkImpl
      extends DatabaseWrapperBase
      implements DatabaseWrapper, Serializable {
            public DatabaseWrapperTOPLinkImpl(Properties appProps, String dataSourceName,
                        String factoryImplTag, String factoryPkgTag);
            public synchronized void beginSession()
                        throws DatabaseException, DatabaseWrapperException;
            public void beginTransaction();
            public void commit()
                        throws DatabaseException, OptimisticLockException,
                        DatabaseWrapperException;
            public void commitAndResume()
                        throws DatabaseException, OptimisticLockException,
```

```
                DatabaseWrapperException;
        public void delete(Object domainObject) throws DatabaseWrapperException;
        public void delete(Vector domainObjects) throws DatabaseWrapperException;
        public void endSession();
        public final ClientSession getCurrentSession();
        public DomainObjectFactory getDomainObjectFactory(Class factoryInterface)
                throws FactoryException;
        public DomainObjectFactory getDomainObjectFactory(String name)
                throws FactoryException;
        public final UnitOfWork getUnitOfWork();
        public final boolean inSession();
        public final boolean inTransaction();
        public Object makePersistent(Object domainObject)
                throws DatabaseWrapperException;
        public Vector makePersistent(Vector domainObjects)
                throws DatabaseWrapperException;
        public void rollback() throws DatabaseWrapperException;
}
public abstract class DomainObjectFactoryTOPLinkImpl extends DomainObjectFactory {
        protected final DatabaseWrapperTOPLinkImpl getDatabaseWrapper();
        protected final void setDatabaseWrapper(DatabaseWrapper databaseWrapper);
}
public class TOPLinkConfigurator {
        public void postLoginConfigureProject(Project project, Properties props)
                throws TOPLinkConfiguratorException;
        public void postLoginConfigureSession(DatabaseSession session, Properties props)
                throws TOPLinkConfiguratorException;
        public void preLoginConfigureProject(Project project, Properties props)
                throws TOPLinkConfiguratorException;
        public void preLoginConfigureSession(DatabaseSession session, Properties props)
                throws TOPLinkConfiguratorException;
}
public final class TOPLinkConstants {
        // Required initialization keys
        final static String USERNAME = "DatabaseUserName";
        final static String PASSWORD = "DatabasePassword";
        final static String PROJECT_SOURCE = "SourceType";
        final static String PROJECT_FILE = "ProjectFile";
        final static String PROJECT_PATH = "ProjectPath";
        final static String PROJECT_CLASS = "ProjectClass";
        final static String CONFIG_CLASS = "ConfigClass";
        // Optional initaillization keys
        final static String OPTIMIZE_DATA_CONVERSION = "Optimize";
        final static String SQL_FLAG = "EnableSQLTrace";
        final static String MAX_RETRY = "MaxRetry";
        final static String MAX_CONNECTIONS = "MaxConnections";
        final static String MIN_CONNECTIONS = "MinConnections";
        public final static String JDBC_DRIVER = "Driver";
        public final static String JDBC_CONNECTION_STRING = "ConnectionString";
}
public class ServerSessionHolder {
        protected ServerSessionHolder();
        public void createServerSession(Properties toplinkProps, String dataSourceName)
                throws DatabaseException, DatabaseWrapperException;
        public ServerSession getServerSession(String dataSourceName)
                throws DatabaseException;
        private  Project initializeProject(Properties dbProps)
                throws DatabaseWrapperException, TOPLinkConfiguratorException;
        public static synchronized ServerSessionHolder instance();
        public void removeServerSession(String dataSourceName)
```

```
                    throws DatabaseException;
            public  void storeServerSession(String dataSourceName, ServerSession session);
}
final public class TOPLinkConfiguratorException extends PersistenceException {
            public TOPLinkConfiguratorException();
            public TOPLinkConfiguratorException(String s);
            public TOPLinkConfiguratorException(String s, Throwable ex);
}


package com.sprint.arch.pageenumeration; (Version 3.5)
public interface PageEnumeration extends EJBObject {
            public Vector next() throws RemoteException;
            public Vector next(DatabaseWrapper dbWrapper) throws RemoteException;
            public boolean nextSucceeds() throws RemoteException;
            public Vector previous() throws RemoteException;
            public Vector previous(DatabaseWrapper dbWrapper) throws RemoteException;
            public boolean previousSucceeds() throws RemoteException;
}
public interface PageEnumerationHome extends EJBHome {
            public PageEnumeration create(Properties appProps, int pageSize,
                        Class factoryInterface, Object[] handleArray)
                        throws CreateException, RemoteException;
            public PageEnumeration create(Properties appProps, int pageSize,
                        String dataSourceName, Class factoryInterface, Object[] handleArray)
                        throws CreateException, RemoteException;
}
public class PageEnumerationImpl implements SessionBean {
            public void ejbActivate() throws RemoteException;
            public void ejbCreate(Properties appProps, int pageSize, Class factoryInterface,
                        Object[] handleArray) throws CreateException, RemoteException;
            public void ejbCreate(Properties appProps, int pageSize, String dataSourceName,
                        Class factoryInterface, Object[] handleArray)
                        throws CreateException, RemoteException;
            public void ejbPassivate() throws RemoteException;
            public void ejbRemove() throws RemoteException;
            private Vector getPage(int pageNumber);
            private Vector getSwizzledPage(int pageNumber);
            private Vector getSwizzledPage(int pageNumber, DatabaseWrapper dbWrapper)
                        throws PersistenceException;
            public Vector next() throws RemoteException;
            public Vector next(DatabaseWrapper dbWrapper) throws RemoteException;
            public boolean nextSucceeds() throws RemoteException;
            public Vector previous() throws RemoteException;
            public Vector previous(DatabaseWrapper dbWrapper) throws RemoteException;
            public boolean previousSucceeds() throws RemoteException;
            public void setSessionContext(SessionContext sessionContext)
                        throws RemoteException;
}
public interface PageableFactory {
public Vector swizzle(Vector handleVector);
}
```

Descriptions

See the JavaDoc Reference for the Java Persistence Framework Release 3.5.

Changes From Release 3.13

- Replaced the use of the Initialization Framework's ApplicationContext object with Java's Properties object. This effectively decouples the Persistence and Initialization frameworks.
- Renamed the collection package to the pageenumeration package.